Developing a Mechanism to Study Code Trustworthiness

Charles Walter, Rose F. Gamble
University of Tulsa, Tulsa OK
charlie-walter@utulsa.edu
gamble@utulsa.edu

Gene M. Alarcon
Air Force Research Laboratory
Wright Patterson AFB
gene.alarcon.1@us.af.mil

Sarah A. Jessup, Chris S. Calhoun
CSRA, Dayton, OH
sarah.jessup@csra.com,
chris.calhoun@csra.com

Abstract

When software code is acquired from a third party or version control repository, programmers assign a level of trust to the code. This trust prompts them to use the code as-is, make minor changes, or rewrite it, which can increase costs and delay deployment. This paper discusses types of degradations to code based on readability and organization expectations and how to present that code as part of a study on programmer trust. Degradations were applied to sixteen of eighteen Java classes that were labeled as acquired from reputable or unknown sources. In a pilot study, participants were asked to determine a level of trustworthiness and whether they would use the code without changes. The results of the pilot study are presented to provide a baseline for the continuance of the study to a larger set of participants and to make adjustments to the presentation environment to improve user experience.

1. Introduction

A programmer's trust in another's code, that is, code that the programmer did not write, is an important but often overlooked part of software projects. Misplaced suspicion can incur additional software development time and cost with programmers rewriting code that already performs correctly and meets requirements, as well as cause programmers to doubt and focus their debugging on code they use but do not trust. In addition to wasted development time, during rewrite programmers can introduce their own bugs.

The issues with a lack of trust extend beyond code that is written by individuals, in-house teams, or third-party vendors. Machine generated code can also be perceived as untrustworthy if it is incompatible with programmer expectations, leading to disapproval for its use. Since machines are increasingly relied on for code generation, programmers must ensure the codes meets requirements, can be reused in different environments, and can be maintained, without being sidetracked due to their distrust of the manner in which the code was written. This perception is problematic as future machines may be tasked to autonomously adapt their code to certain situations. If code must go through a certification process, for example to meet security requirements, delays in redeployment can be exacerbated if the machine generated code must be rewritten due to mistrust. We propose that if human and machine-generated code adheres to a set of coding styles that are expected by intermediate and expert programmers of the language used, it would improve its trustworthiness. Ideally, this would lead to a greater trust in code given to contractors or received by companies, preventing programmers from losing time "fixing" working code and potentially allowing machine-written code to be as trusted as a human-written version.

This paper examines an initial set of factors to determine their relationship to programmer trust in code written by someone else. Two of the factors, *readability* and *organization*, are the first in a series of factors to be studied that point to specific ways working code can be degraded to potentially decrease trustworthiness in its incorporation or use by a software developer. These factors were identified using a cognitive task analysis (CTA) as described in [1]. Using a web-based platform, eighteen (18) Java classes are presented as images to study participant responses. In addition to their degradations, each Java class is labeled as coming from a reputable or unknown source. Participants are asked to rate the trustworthiness of the code and determine if they would use the code without changes. The main research questions for the study are:

- RQ1: Does the readability of code affect its trustworthiness?
- RQ2: Does the organization of code affect its trustworthiness?
- RQ3: Does basic knowledge of the source of the code (i.e. reputable vs. unknown) affect its trustworthiness?
- RQ4: Is the trustworthiness rating of the code related to whether a programmer would or would not use the code?

URI: http://hdl.handle.net/10125/41864
ISBN: 978-0-9981331-0-2

In this paper, we overview the platform created for the study. We detail finer-grained degradations, along with providing examples of each, and how they are dispersed throughout the code artifacts to designate them as low, medium, or high readability or organization. We discuss the results of a pilot study of 12 participants, which provided foresight into the potential results of the full study planned for 72 participants. The pilot study also provided an understanding of usability of the platform, whether the image-based interface was appropriate for code trustworthiness assessment, and what the average time was to complete the study.

2. Background

There are few studies regarding why programmers trust some code over others. Kelly and Shepard [7] looked at the number of coding errors found in software inspections when those inspections were performed individually versus those performed by a group. Their findings indicated that interacting groups detected fewer new issues and rejected errors detected individually. Their study showed a higher likelihood of increased trust in external code when a group review is performed over the trust in the same external code given by a single reviewer.

Rigby and Bird [12] discussed the usefulness of the software review process. They focused on the benefits of finding errors and discussing potential solutions in open source code. Because open source code is widely trusted by its users, they presented a good example of how discussion can lead to greater trust in code that is written by others. By looking at open source projects with many users, it is possible to see examples of trusted code written by others. Thus, the acceptance of open source code can lead to an increase in the reputation of the programmer(s) who crafted it.

When a programmer is forced to maintain code with defects, Albayrak and Davenport [2] determined that defects in the formatting of the code increases the false positive rate and lowers the number of functional defects detected. This study implied that non-logical defects, such as the way the code or its comments are formatted, can lead to a mistrust of the code itself, regardless of whether the code is logically correct.

Naedele and Koch [10] examined a method of ensuring trust in code after it has been transferred to another system for review by another program. The authors focused on how ensuring the delivery of tamper-proof code, i.e. nothing happens to the code in transit, along with the reputation and liability of the supplier of the code, can determine overall trust. While this focus is important in understanding trust decisions, it treats the code as a black box, preventing the code itself from being the basis of the trust decision.

When examining software inspections, Porter, et al. [11] identified one of the causes of variation in the outcome of the inspection as Code Unit Factors. These factors include the author, the size of the code, when the code was written, and the functionality of the code. The authors showed that these are major contributors to the number of defects associated with the code and, thus, should be further examined as potential trust markers.

Kopec et al [8] showed that intermediate-level programming students can make drastic mistakes on even simple code. Using simple examples, the authors examined multiple correct and incorrect methods of solving the same programming problem. The differences among the resulting code implied programmers do not write their code in exactly the same way. The study indicated the possibility that programmers may be less likely to understand and, by extension, trust, code that is unlike the code they write.

The readability of code has been previously studied, though not from a perspective of trustworthiness. Tashtoush et al. [14] defined a formula to automatically analyze the readability of simple Java code. They used online surveys to establish individual weights for each feature, then tested the readability of code samples with those features to fine-tune their algorithm. They found that some features, such as meaningful variable names and consistency, raised the overall readability of the code samples, while others, such as recursive functions, nested loops, and arithmetic formulas, lowered the overall readability. As some algorithms cannot be written without the use of recursion or nested loops, it is important to understand the factors that can be adjusted to ensure that code samples which include these features are still readable.

3. Readability and Organization Degradations

For this study, we examined detailed degradations of readability and organization, along with a simple distinction between the code source of reputable or unknown. These three factors were identified by a cognitive task analysis associated with the study [1]. The factors were identified as those that led to greater transparency in the code, which is believed to increase its trustworthiness. Readability is defined as the ease with which a programmer or analyst can review the code and understand its intent. Organization is defined as the manner in which the control structure and logic of the code is represented and understandable.

We targeted Java classes for the study as it is one of the more popular programming languages. Thus, readability and organization qualities were derived from Java Style Guides [5, 6, 13], an extensive search of questions and answers on stackoverflow.com, and a commonly used undergraduate textbook [4] for Java coding standards and common practices.

Table 1 lists the readability degradations that were imposed on the code. Misuse of case is segregated into the different entities where the wrong case used in the name could signal a novice programmer. Misuse of braces can impact readability because brace usage stems from early training on Java convention.

In some languages proper indentation is required, so high skilled programmers maintain proper indentation even when it is not needed for accurate code execution. The last readability degradation points to line length and line wrapping. How long a line is and how blank lines are managed can point to programmers that are unconcerned about their code being read by others. Along with improper use, inconsistent use of accepted conventions can indicate poor training of an individual or group of programmers.

Table 2 lists the organization degradations that were imposed on the code. These degradations focused on the structural manifestation of the code and highlighted the programmer's mindset and training. For example, how a programmer groups methods, including those that are overloaded, may indicate how the code was derived initially and later revised.

Table 1. Readability Degradations

1. Misuse of case	a) For packages
	b) For classes and interfaces
	c) For methods and variables
	d) For constants
2. Misuse of braces	a) Line break before an opening brace
	b) No line break after an opening brace
	c) No line break before a closing brace
	d) Line break after a brace that precedes an else
	e) Missing a space before an opening or closing brace
3. Misuse of indentation	a) Improper indentation given code position
	b) Inconsistent indentation
4. Improper line length and line wrapping	a) Unnecessarily exceeds character limit without wrapping
	b) Missing blank lines to indicate logical grouping
	c) Use of too many and unnecessary blank lines

Table 2. Organization Degradations

1. Poor grouping of methods	a) Any form
2. Misuse of declarations	a) Import statements used improperly
	b) More than one variable per line
	c) Variables not initialized as soon as possible
	d) Overuse of public instance and class variables
3. Ambiguous control flow	a) Improper, unnecessary, or confusing use of "break" or "continue"
	b) Unnecessary or confusing nesting of blocks
	c) Multiple function calls or unnecessarily grouping block on one line
	d) Switch statement does not have a default case
	e) Switch statement with no "break" does not comment explicit continuation to next statement group
4. Improper exception handling	a) Any form
5. Statements unnecessarily require additional review	a) Compressed if statements
	b) Unusual return statements
	c) Multiple classes
	d) Inconsistent blocks

The misuse of declarations, as described in Table 2, may also indicate code that was revised multiple times with the placement of declarations be placed directly with newly inserted code. Ambiguous control flow, and improper exception handling may point to a programmer creating haphazard code or just being lazy. Statements that may be overly complex or structured in a way that requires deeper analysis may indicate a poor programming style or a careless programmer. Inconsistency of organization characteristics within the same code may indicate that multiple programmers revised the code, which could promote distrust.

A total of 18 code artifacts, i.e. Java classes, for this study, were taken from a variety of sources. Either they could be classified as having existing degradations, or we augmented them with degradations without creating code that did not compile or produce the intended output. Thus, all resulting code artifacts compiles and works as intended. The code was sanitized to prevent the study participant from forming any biases. In addition, the study participants were told that all comments were removed, again to eliminate bias toward commenting styles and practices, which provide different factors for study according to the CTA [1]. Each code artifact was designated as

- coming from a Reputable or Unknown source
- high, medium, or low readability
- high, medium, or low organization

to satisfy all possible combinations.

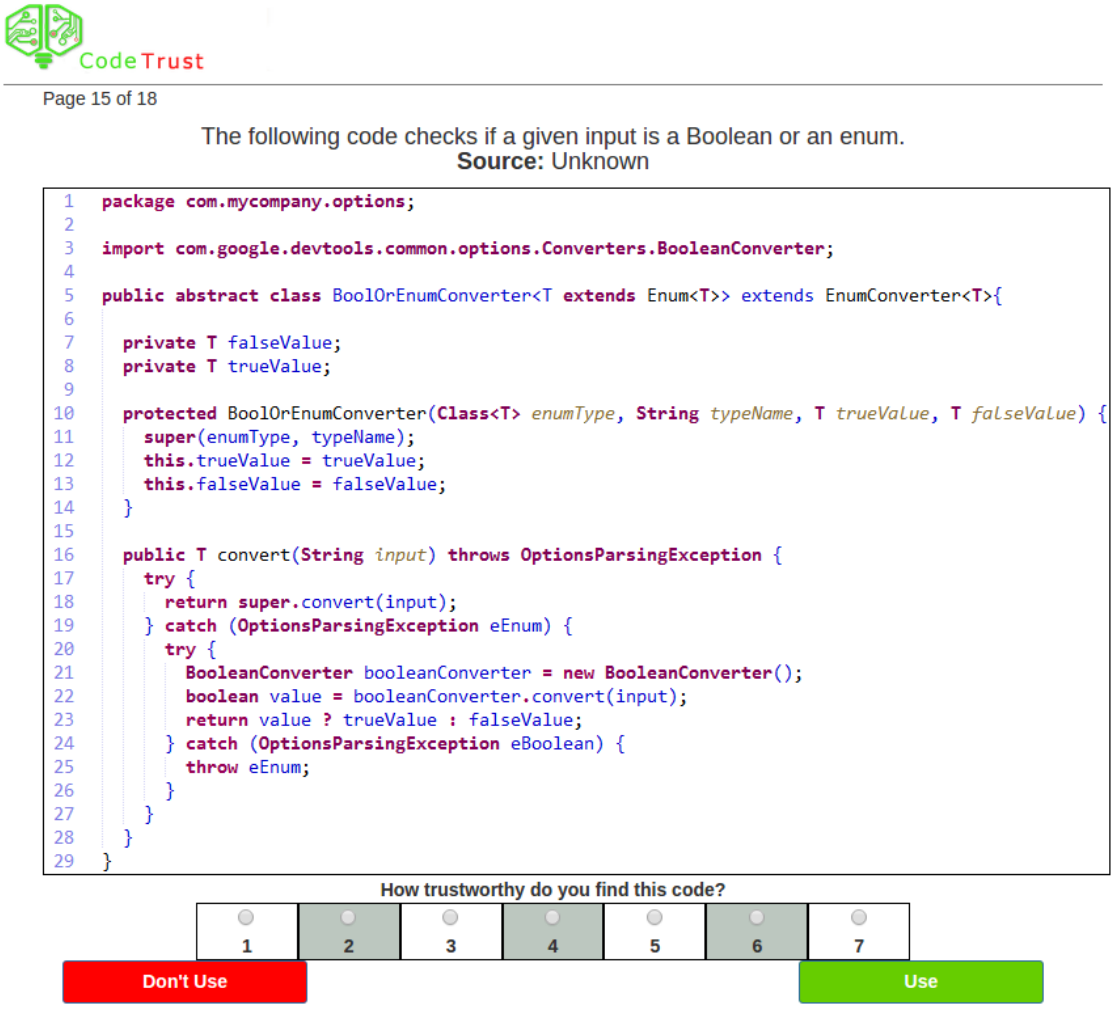

Figure 1. Sample Code Presented to Study Participant

A high readability or organization value implies that style guidelines and best practices are followed within the code. A medium readability or organization value implies that there are multiple instances (generally between 3-7) of the same or different degradations. A low readability or organization value implies that there are a significant of degradations (generally greater than 7 instances) and that there were at least 2 different degradation representations.

Each degraded artifact had a different selection and combination of degradations, in an effort to prevent the code from appearing to be too unnatural or unlike something any coder would write. While the number of degradations provided a metric, their inconsistent appearance and their percentage of representation given the total lines of code also distinguished between medium and low readability or organization. Consistency in the degradation placement in the code was used at medium levels with the understanding that it was the way the programmer was trained (possibly poorly) to write code. Inconsistency in the application of a degradation throughout the code was used at the low levels to potentially indicate that multiple programmers used the code or that a single programmer was careless or unconcerned about the reuse of the code. Each code artifact was analyzed by five subject matter experts independently from two different organizations to ensure that it met the assigned degradation level.

4. Study Platform

In order to present the code to study participants for review and a decision on its trustworthiness, we constructed a web application platform that allowed the study to be administered in multiple cities without loss of data. The platform was created in Ember, a javascript framework allowing for minimal communication with a server and for all data to be stored in the browser until the completion of the study.

Given that the expected participants needed to have three years of coding experience and familiarity with Java, they would examine code using an editor (with color coding) or an IDE, such as Eclipse. Such programmers may also search the code, run a code inspection tool on it, and see updates by other team members, as well as compile and execute it. These considerations complicated the presentation of the information, because every programmer is different and simulating one's environment or process would not necessarily be engaging to another programmer. We experimented with presenting a set of images of a single Java class that included the class in a standard editor with color coding, the result of an inspection tool, and the result of a "diff" command to show differences in versions. Since the only common artifact that was acceptable was just the code presentation image, we opted for that in the study.

Each artifact was on its own page with a general description of what the class was intended to do at the top of the page, along with the source. Figure 1 shows a sample page in the study.

Figure 2 – Figure 5 provide samples of degradations. Figure 2 shows multiple readability (R) degradations to achieve a low readability level. Line 83 has a line break before an opening brace (R2.a). Improper indentation given code position (R3.a) and inconsistent indentation (R3.b) appear on lines 85 and 86. Line 88 has no line break before a closing brace (R2.c) and is missing a space before a closing brace (R2.e).

Figure 3 shows multiple organization (O) degradations to achieve a low organization level. Lines 66-68 have a switch statement with no default case (O3.d) and which has no "break" but does not comment explicit continuation to next statement group (O3.e) exhibiting ambiguous control flow. Lines 69-71 displays improper exception handling (O4.a).

Figure 4 shows an example of combining readability and organization degradations. It has a line break before an opening brace (R2.a) and no line break after an opening brace (R2.b) on line 44. It also has an overuse of public instance and class variables (O2.d) on lines 38-41. These degradations combine with other in this code artifact to have a low readability and a low organization.

Figure 5 shows a second example of the misuse of case for methods and variables (R1.c) on line 37, a line break before an opening brace (R2.a) on line 38, and a compressed if statement requiring more in depth review (O5.a) on line 39 in a portion of a code artifact that exhibits medium readability and medium organization.

```
82          if (errors != null && errors.size() > 0)
83          {
84              message.append(':');
85                  for (FieldError fieldError : errors)
86                  message.append(' ').append(format(fieldError)).append(',');
87              message.deleteCharAt(message.length() - 1);}
88      return message.toString();}}
```

Figure 2. Sample Readability Degradations

```
65          try{
66            switch (convertView){
67              view = (ViewGroup) convertView;
68            }
69          } catch (NullPointerException e){
70            view = (ViewGroup) mInflater.inflate(R.layout.rounded_item, parent, false);
71          }
```

Figure 3. Sample Organization Degradations

```
38      public final int mResId;
39      public final String mLine1;
40      public final String mLine2;
41      public final ScaleType mScaleType;
42
43      ColorItem(int resid, String line1, String line2, ScaleType scaleType)
44      { mResId = resid;
45        mLine1 = line1;
46        mLine2 = line2;
47        mScaleType = scaleType;
48      }
49    }
```

Figure 4. Combined Readability and Organization Degradations (#1)

```
37      char Consume()
38      {
39          char val = pos >= length ? eoF : input[pos];
40          pos++;
41          return val;
42      }
```

Figure 5. Combined Readability and Organization Degradations (#2)

5. The Pilot Study

For inclusion in the pilot study participants were required to have at least 3 years of experience in computer programming and be a competent Java programmer. Pilot study participants were recruited from local industry and from The University of Tulsa computer science graduate students. All participants met the requirements of having at least 3 years of programming experience and a working knowledge of Java. A total of 12 participants (11 males and 1 female) with a mean age of 25.5 years and a SD of 7.5 were recruited for the initial experiment. These participants were not compensated. The age range was 21 to 48. Eight participants had completed a 4-year degree, 2 had completed a graduate degree, and 2 had less than 4 years of college.

At the start of the study, a user answers demographic questions and self-report surveys which include a Mayer-Davis Propensity to Trust Scale [9], a mini IPIP [3], and a series of Suspicion Propensity Index (SPI) situational-based items. The participants

were then informed of the number of code artifacts they will be reviewing, that there were purposely no comments included in the artifacts, and that they were reviewing the code only to decide if they would use the code in a project that had need of the functions the code claimed it could perform. Participants were told that they must decide if they will use or not use the code as it is written. In addition, they were asked to rate how trustworthy they found the code using a 7-point Likert scale as shown in Figure 1. Participants could ask clarifying questions to study proctors about the code artifacts and the operation of the platform.

5.1. Data Collection

The platform collected data from the user as decisions were made. Code artifacts were shown to the user one at a time with a description of what the code does and a source, either reputable or unknown, for context. After reviewing the code, a user rated the trustworthiness and then clicked "Use" or "Don't Use" (see Figure 1) If a user clicked "Use," the platform directed them to the next code artifact without asking for feedback, as the user deemed the code trustworthy. If a user clicked "Don't Use," an additional dialog box appeared that asked for comments on why the code would not be used, allowing for more detailed feedback on negative answers. After inserting comments, the user was then able to click submit, which directed them to next artifact.

For each content item, a database retained its rating, trust decision, and explanation of mistrust against a user ID. If a user attempted to move forward in the study without selecting a trust rating, the system responded with a request to choose a rating level before continuing. To ensure that a user could exit the study at any time without any personal information being collected, all data was stored locally in the browser until the completion of the study.

5.2. Evaluation

To address RQ1-RQ3, we analyzed the data using three univariate ANOVAs. ANOVA is a collection of statistical tools for analyzing differences between multiple group means. We analyzed the data with a null hypothesis of no significant differences among manipulations of code. If the null hypothesis was rejected, we applied post hoc Bonferroni analysis to study the differences among code manipulations. All the results are reported on the basis of an alpha level of 0.05. ANOVA results illustrate significant main effects of readability ($F(2,216) = 8.704$, $p<0.001$), organization ($F(2,216) = 3.306$, $p=0.039$), and source ($F(1,214) = 19.526$, $p<0.001$). All factors resulted in a critical p value less than the selected significance level, indicating the trustworthiness scores differ significantly across degradation groups. The Bonferroni post hoc analysis was used to contrast multiple comparisons to determine which mean differences are significantly different from each other as discussed below.

Analysis of the readability condition indicates high readability was significantly different from medium and low readability, as indicated in Figure 6. High readability led to higher perceptions of trustworthiness in the code, but once degraded there were no statistically significant differences in perceptions of trustworthiness. The organization condition indicates high organization of the code was significantly different from medium and low organization, as shown in Figure 7. However, once code was degraded it was perceived as more trustworthy than in the high organization condition. Lastly, there was significant difference between reputable and unknown sources of code, as depicted in Figure 8. If the code was said to be reputable it was perceived as more trustworthy than code from an unknown source.

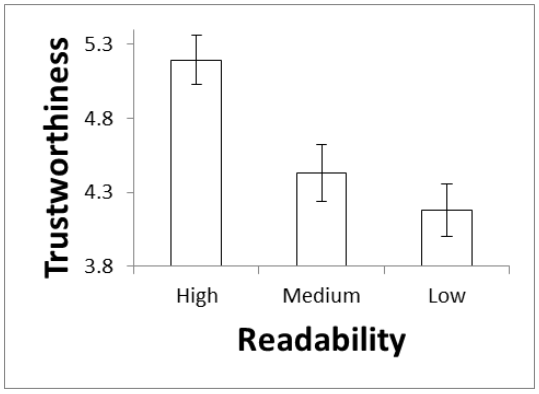

Figure 6. Readability Analysis

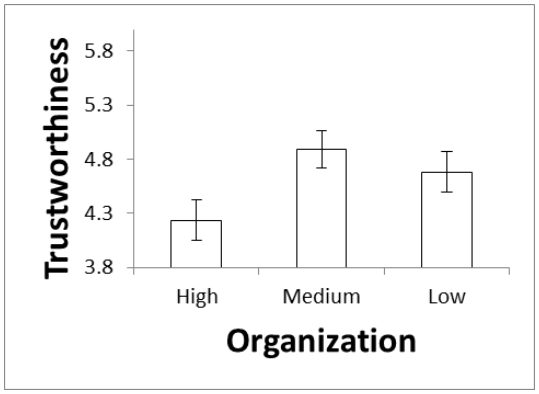

Figure 7. Organization Analysis

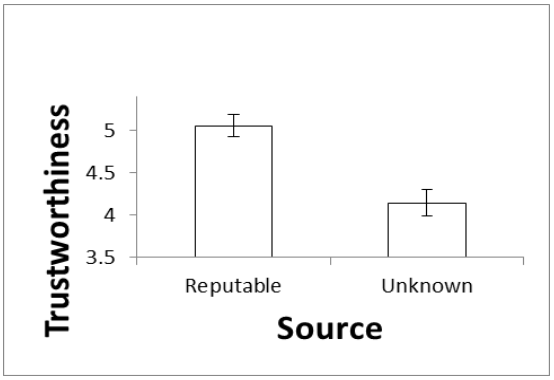

Figure 8. Source Analysis

Table 3 shows the Use/Don't Use selections given the artifacts classification for readability and organization.

To address RQ4, a logistic regression was performed to ascertain the effects of readability, organization and source on the likelihood that participants would use the code. The logistic regression model was significant ($X2$ (7) = 18.067, $p<.01$). The model explained 11% of the variance in the decision to use the code and correctly classified 65.7% of the cases. Medium readability code was 0.34 times less likely to be used, and low readability code was 0.38 times less likely to be used than high readability code. Low organization code was 2.31 times more likely to be used than high organization code. There was no difference between medium and low organization. Code that was from an unknown source was 0.595 times less likely to be used than code from a reputable source.

To better understand why there was a difference in trusting organization degradations and if this could propagate to the full study, we logged how many times a participant trusted code that had a particular degradation. We totaled the number of "don't use" decisions for artifacts containing a particular degradation type and divided by the number of artifacts where that degradation type appeared. Dividing that result by the 12 participants yielded the histogram in Figure 9, representing the percentage of time a degradation was distrusted when it appeared in a code artifact, or strength of the distrust with respect to all degradations.

Table 3. Pilot Study "Use" and "Don't Use" Choices for Code Artifacts given their Classifications

			Readability					
			High		Medium		Low	
			Use	Don't Use	Use	Don't Use	Use	Don't Use
Organization	High	Unknown	8	4	5	7	5	7
		Reputable	10	2	6	6	5	7
	Medium	Unknown	7	5	8	4	7	5
		Reputable	11	1	10	2	9	3
	Low	Unknown	10	2	6	6	7	5
		Reputable	10	2	10	2	9	3

It is visually apparent that that organization degradations have lower levels of distrust as compared to the readability degradations. The average strength of distrust over the readability degradations is 0.43 versus an average of 0.27 for organization degradations. It should be noted that there are 53 appearances of readability degradations across the 18 code artifacts versus 38 appearances of organization degradations. Thus, it is possible that the organization degradations were not as apparent as the readability degradations. However, it does not answer the question of why high organization caused distrust overall even when readability was low (see also Table 3). Perhaps these structural degradations are common even though they are not considered best practices, but are coded in this manner for expediency. If Java programmers are unconcerned about organization, then it may be suspect if the code is too structured, potentially indicating a novice programmer trying to be very careful.

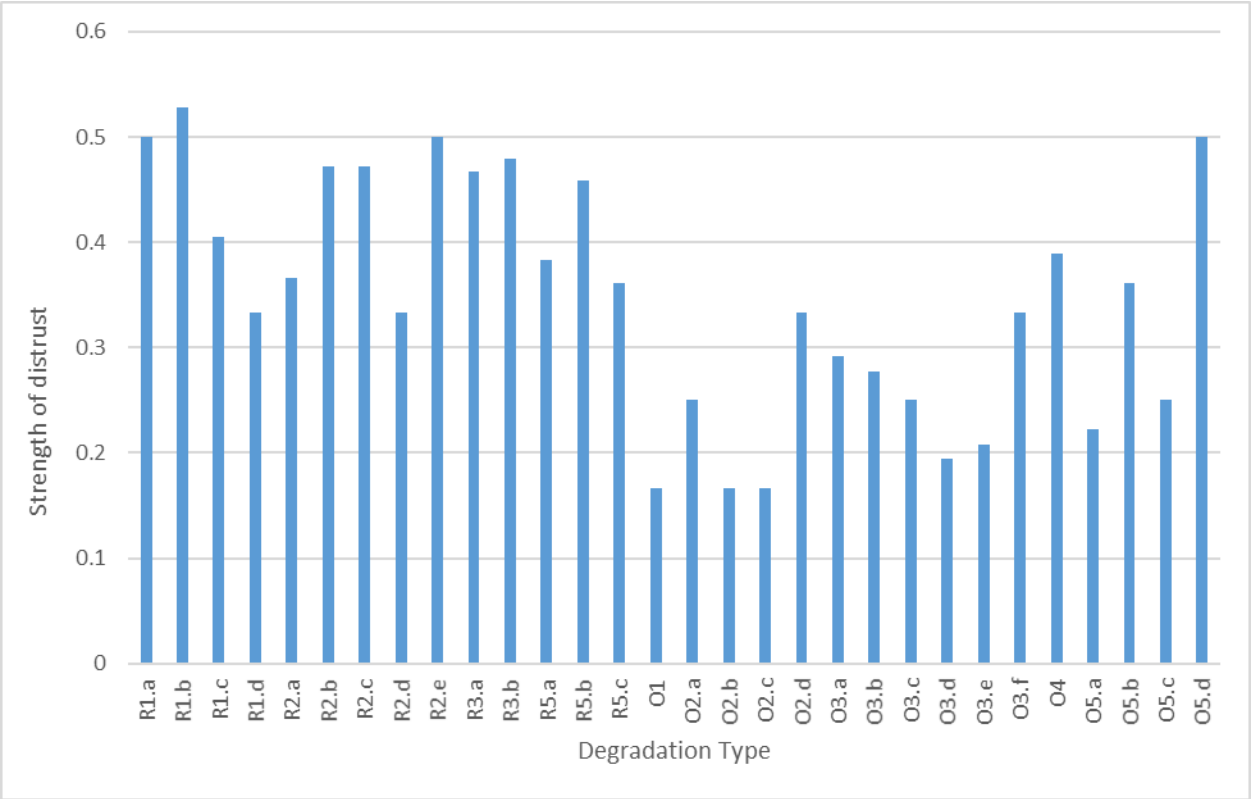

Figure 9. Percentage of Time a Degradation was Distrusted when it Appeared in a Code Artifact

6. Discussion and Conclusion

In addition to the initial readability, organization, and source analyses, the pilot study provided insight into how the platform could be refined to improve both analysis understanding and user experience. For analysis understanding, allowing commenting on why a programmer would use the code might point to why certain organization degradations were trusted. In fact, some participants commented at the end of the study that they wished to explain their choices when they would trust the code. The results of the pilot study are encouraging with respect to readability and source. Organization degradations may need to be revisited if the full study has a similar analysis.

The full study of a larger set participants is underway. These participants are compensated. More detailed instructions are given at the start of the study and the code artifacts have not been changed. The pilot study participants were timed only from start to finish, but the full study has timings associated with each code artifact to provide insight into whether degraded code is more quickly detectable. To improve user experience, a discussion of the code coloration is provided prior to the start of the study. The images used a particular SublimeText Theme that results in some unexpected text colors requiring users to ask for clarification on specific sections of the code.

Our future effort will expand the analysis to examine the degradations more closely with the larger sample size, as well as look at the decision times for each artifact and its relationship to the degradations. Additionally, we will further investigate the effect of comments within the code and how it relates to perceived code trustworthiness. The plan is to continue the study with additional forms of degradation as found in the CTA [1] to develop an understanding of coding styles that are commonly mistrusted. Ideally, this could lead to greater trust in code given to contractors or acquired by companies, preventing programmers from losing time “fixing” working code and potentially allowing machine-written code to be as trusted as a human-written version.

Acknowledgement. This research was funded in part by the Air Force Research Laboratory (Contract FA8650-09-D-6939/0033). The findings and conclusions in this report are those of the authors and do not necessarily represent the official position of the Air Force.

7. References

[1] G. M. Alarcon, L. G. Militello, P. Ryan, S. A. Jessup, C. S. Calhoun, and J. B. Lyons, “A descriptive model of computer code trustworthiness.” *Journal of Cognitive Engineering and Decision Making*, (in press), 2016, online as doi:10.1177/1555343416657236.

[2] Ö. Albayrak and D. Davenport, "Impact of Maintainability defects on Code Inspections," *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* - ESEM'10, 2010.

[3] M. B. Donnellan, F. L. Oswald, B. M. Baird, and R. E. Lucas, "The Mini-IPIP Scales: Tiny-Yet-Effective Measures of the Big Five Factors of Personality," *Psychological Assessment*, vol. 18, 2006, pp. 192-203.

[4] T. Gaddis, *Starting Out with Java: From Control Structures Through Objects*, Pearson, Addison-Wesley, 2015.

[5] "Geotechnical Software Services, Java Programming Style Guidelines, http://geosoft.no/development/javastyle.html," 2015.

[6] "Google, Java Style Guidelines, https://google.github.io/styleguide/javaguide.html," 2014.

[7] D. Kelly and T. Shepard, "An experiment to investigate interacting versus nominal groups in software inspection," *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research*, 2003, pp. 122-134.

[8] D. Kopec, G. Yarmish, and P. Cheung, "A Description and Study of Intermediate Student Programmer Errors," *SIGCSE Bulletin*, vol. 39, 2007, pp. 146-156.

[9] R. C. Mayer and J. H. Davis, "The Effect of Performance Appraisal System on Trust for Management: A Field Quasi-Experiment," *Journal of Applied Psychology*, vol. 84, 1999, pp. 123-136.

[10] M. Naedele and T. E. Koch, "Trust and Tamper-Proof Software Delivery," *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems* - SESS '06, 2006, pp. 51-57.

[11] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the Sources of Variation in Software Inspections," *ACM Transactions on Software Engineering and Methodology*, vol. 7, 1998, pp. 41-79.

[12] P. C. Rigby and C. Bird, "Convergent Contemporary Software Peer Review Practices," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* - ESEC/FSE 2013, 2013, pp. 202-212.

[13] "Sun Microsystems, Java Code Conventions, http://www.oracle.com/technetwork/java/codeconventions-150003.pdf," 1997.

[14] Y. Tashtoush, Z. Odat, I. Alsmadi, and M. Yatim. "Impact of Programming Features on Code Readability." *International Journal of Software Engineering and Its Applications*, 2013, pp. 441-458.